

# Continuous Detour Queries in Spatial Networks

Sarana Nutanong\*, Egemen Tanin<sup>†‡</sup>, Jie Shao<sup>†</sup>, Rui Zhang<sup>†</sup>, Kotagiri Ramamohanarao<sup>†‡</sup>

\*Inst. for Advanced Computer Studies  
University of Maryland,  
College Park, MD  
{nutanong}@umiacs.umd.edu

<sup>†</sup>Dept. of Computer Science and Software Engineering  
University of Melbourne, Victoria, Australia  
{egemen,jsh,rui,rao}@csse.unimelb.edu.au  
<sup>‡</sup>NICTA Victoria Research Laboratory, Australia



**Abstract**—We study the problem of finding the shortest route between two locations that includes a stopover of a given type. An example scenario of this problem is given as follows: “On the way to Bob’s place, Alice searches for a nearby take-away Italian restaurant to buy a pizza.” Assuming that Alice is interested in minimizing the total trip distance, this scenario can be modelled as a query where the current Alice’s location (start) and Bob’s place (destination) function as query points. Based on these two query points, we find the *minimum detour object (MDO)*, i.e., a stopover that minimizes the sum of the distances: (i) from the start to the stopover, and (ii) from the stopover to the destination. In a realistic location-based application environment, a user can be indecisive about committing to a particular detour option. The user may wish to browse multiple ( $k$ ) MDOs before making a decision. Furthermore, when a user moves, the  $k$ MDO results at one location may become obsolete. We propose a method for *continuous detour query (CDQ)* processing based on incremental construction of a *shortest path tree*. We conducted experimental studies to compare the performance of our proposed method against two methods derived from existing  $k$ -nearest neighbor querying techniques using real road-network datasets. Experimental results show that our proposed method significantly outperforms the two competitive techniques.

**Index Terms**—Continuous queries, spatial network, spatial databases

## 1 INTRODUCTION

Location-based services allow users to search nearby facilities and find the shortest route between two locations. In many cases, a user, who is travelling to a specific destination, may be interested in finding a stopover that does not introduce significant costs to the trip. An application scenario can be given as: “On the way to Bob’s place, Alice searches for a nearby take-away Italian restaurant to buy a pizza.” One approach to addressing the problem in this scenario is: (1) to calculate the shortest path from Alice’s location  $q_s$  to Bob’s place  $q_e$ ; (2) to find the restaurant nearest to that path [4], [27]. In Figure 1, the shortest path from  $q_s$  to  $q_e$  is highlighted in grey. Objects  $a$ ,  $b$ , and  $c$  represent restaurants that satisfy the search criteria. The object nearest to the shortest path is  $b$ , which incurs a deviation of 4 units from the shortest path and an overall trip distance of 21 units. Although this approach is aimed at finding the object residing closest to the route, we argue that it does not necessarily produce the overall shortest path. This is because, this approach uses the deviation from a precomputed shortest path rather than the overall trip distance.

To address this problem, we formulate a new query type, called the *detour query*, which uses the *trip distance* as the optimization measure. Given a set  $\mathcal{D}$  of detour objects (stopovers), a starting location  $q_s$  and an end location  $q_e$ , the detour query returns a *minimum detour object (MDO)*. An MDO is an object  $p$  in  $\mathcal{D}$  that minimizes the TRIPDIST (the sum of: (i) the distance from  $q_s$  to  $p$ , and (ii) the distance from  $p$  to  $q_e$ ). Figure 1 illustrates an example of the detour query where the user wishes to travel from  $q_s$  to  $q_e$ , the MDO in this scenario is  $a$ , which provides the TRIPDIST of 15 units (6 units shorter than the solution from the previous approach).

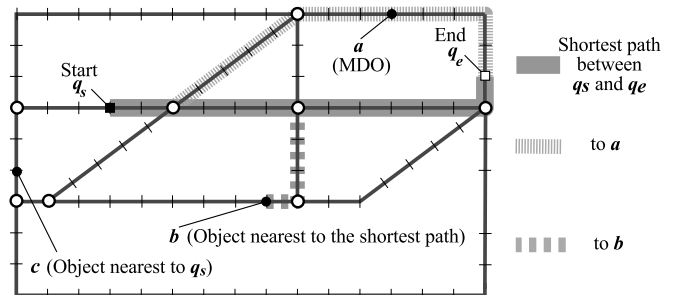


Fig. 1. Minimum detour object with respect to a starting point  $q_s$  and an end point  $q_e$

In order to provide support for realistic location-based applications, we tackle the aforementioned problem from two aspects. The first aspect is the ability to browse and to compare multiple results. By displaying  $k$  MDOs at a time, we allow a user to browse and to select the option with which they are most satisfied. In Figure 1, for example, the 2 MDOs with respect to  $q_s$  and  $q_e$  are  $\langle a, b \rangle$ . Although  $a$  provides the smallest TRIPDIST,  $b$  can be more appealing in terms of food quality or prices.

The second aspect is continuous monitoring of  $k$  MDOs. Ongoing  $k$ MDO monitoring provides users who wish to take time browsing query results with up-to-date information in the same manner as other continuous spatial queries [3], [4], [28]. A user may browse to gather information without an urgent need to commit to a particular decision [1]. As a result, browsing users usually take more time to make decisions than users who are searching for something specific [1], [2], [14], [25]. An application scenario where browsing applies is given as “on the way to Bob’s party, Alice is consulting Bob whether

she should drop by an Italian restaurant and purchase take away pizzas to bring to the party”. Making a detour decision in this scenario can be an ongoing process. It involves Alice and Bob collaboratively browsing possible detour options. Using CDQ, Alice could obtain results and discuss them with Bob. CDQ also allows Alice to inform Bob when results change and ensures that they always have up-to-date information to help make a detour decision whenever they are happy.

A straightforward approach to solving the *continuous detour query (CDQ)* problem is to evaluate the  $k$ MDO at each intersection along the trajectory. Specifically, we may consider the TRIPDIST,  $(\text{DIST}(\mathbf{q}_s, \mathbf{p}) + \text{DIST}(\mathbf{p}, \mathbf{q}_e))$ , as an aggregate distance function [19] with two query points  $\mathbf{q}_s$  and  $\mathbf{q}_e$ . Then we search for  $k$  objects with the minimum TRIPDISTs for the give locations of  $\mathbf{q}_s$  and  $\mathbf{q}_e$ . Since the starting location  $\mathbf{q}_s$  changes over time, this approach incurs repetitive evaluation of network distances. We use this method as one of the comparators in our experiments.

In this paper, we propose a CDQ solution which incrementally evaluates the  $k$ MDO results at different intersections according to the TRIPDIST measure. Consequently, repetitive distance evaluation is avoided.

The contributions of this paper are summarized as follows.

- We formalize the problem of continuous evaluation of  $k$  MDOs for a moving query and a fixed destination in a spatial network.
- We propose a novel solution to the CDQ problem that: (i) is capable of handling multiple MDOs, (ii) does not require access to all data objects, and (iii) does not incur repetitive distance evaluation as the query point moves.
- We conduct extensive experiments with a real road network dataset and realistic application scenarios. Our experimental results show that our proposed method significantly outperforms a competitive method derived from the *aggregate  $k$  nearest neighbor (AkNN)* querying domain [19], [26].

The rest of the paper is organized as follows. Section 2 presents the problem setting. In Section 3, we discuss related work, i.e., nearest neighbor (NN) queries in spatial networks and route planning queries. Section 4 presents background knowledge on the Dijkstra’s algorithm and Voronoi diagrams which will be needed for our explanations. In Section 5, we describe our proposed solution. A competitive solution and discussion are presented in Sections 6 and 7, respectively. In Section 8, we report experimental results. This paper is concluded in Section 9.

## 2 PROBLEM FORMULATION

A spatial network  $G(N, E)$  is represented by a set  $N$  of nodes (intersections) and a set  $E$  of edges (road segments). For any given two points  $\mathbf{p}_1$  and  $\mathbf{p}_2$  on  $G(N, E)$ , the distance  $\text{DIST}(\mathbf{p}_1, \mathbf{p}_2)$  is the distance via the shortest path from  $\mathbf{p}_1$  to  $\mathbf{p}_2$ . For ease of exposition, we use undirected graphs in our presentation, which means that  $G(N, E)$  satisfies *symmetry* and *triangle inequality* among the other metric space condi-

tions.<sup>1</sup> Notations frequently used in this paper are summarized in Table 1.

TABLE 1  
Frequently used notations

Notation	Meaning
$G(N, E)$	Spatial network.
$\text{EDGE}(\mathbf{n}_i, \mathbf{n}_j)$	Edge that connects $\mathbf{n}_i$ to $\mathbf{n}_j$ .
$\mathcal{D}$	Set of data objects represented as points.
$\mathbf{q}$	Moving query point.
$\mathbf{q}_s$	Start location.
$\mathbf{q}_e$	End location.
$\text{PATH}(\mathbf{p}_1, \mathbf{p}_2)$	Shortest path from $\mathbf{p}_1$ to $\mathbf{p}_2$ .
$\text{DIST}(\mathbf{p}_1, \mathbf{p}_2)$	Length of $\text{PATH}(\mathbf{p}_1, \mathbf{p}_2)$ .
$\text{TRIPDIST}(\mathbf{q}_s, \mathbf{p}, \mathbf{q}_e)$	$\text{DIST}(\mathbf{q}_s, \mathbf{p}) + \text{DIST}(\mathbf{p}, \mathbf{q}_e)$ .
MDO	Minimum detour object.
$k$	Number of resultant objects requested.
$k$ MDO	List of $k$ MDOs ranked using TRIPDIST.

We first define the detour query as a query that returns the MDO, and also give the definition of MDO as follows.

*Definition 1 (Minimum Detour Object (MDO)):* Given a set  $\mathcal{D}$  of data objects in a spatial network  $G(N, E)$ , the MDO is the object  $\mathbf{p}$  in  $\mathcal{D}$  with the smallest  $\text{TRIPDIST}(\mathbf{q}_s, \mathbf{p}, \mathbf{q}_e)$ , where  $\mathbf{q}_s$  and  $\mathbf{q}_e$  represent the coordinates of start and end locations, respectively.

Based on this MDO definition, the  $k$ MDO is defined as a sorted list of  $k$  detour objects with the smallest TRIPDISTs.

Second, our proposed query is defined as follows.

*Definition 2 (Continuous Detour Query (CDQ)):* Given a set  $\mathcal{D}$  of data objects, a moving query point  $\mathbf{q}$  and a destination  $\mathbf{q}_e$  in  $G(N, E)$ , the CDQ query continuously finds the  $k$ MDO with respect to a  $\mathbf{q}$  and  $\mathbf{q}_e$ .

## 3 RELATED WORK

### 3.1 NN Queries in Spatial Networks

Papadias et al. [20] introduced two frameworks for the spatial-network  $k$ NN query: the *incremental euclidean restriction (IER)* and *incremental network expansion (INE)*. IER applies the property that the Euclidean distance between any two network nodes is a lower bound of their network distance to prune the search space. INE performs network expansion similar to the Dijkstra’s algorithm [9] from the query point and examines data objects in the order they are encountered. They showed that INE performs better than IER in general.

As an optimization of IER, Deng et al. [8] proposed incremental *lower bound constraint (LBC)*. The LBC method calculates distance lower bounds of objects for pruning purposes. Hence, the workload from network distance calculation is greatly reduced.

Kolahdouzan and Shahabi [10] presented a *Voronoi-based network nearest neighbor (VN<sup>3</sup>)* approach to evaluate the  $k$ NN query by decomposing the data space using the first-order network Voronoi diagram with respect to data objects. Finding the  $k$  NNs of a query point  $\mathbf{q}$  is done by: (i) identifying the first

1. In the formulation of our algorithms, however, the distance function  $\text{DIST}(\cdot, \cdot)$ , as well as, the notations  $\text{EDGE}(\cdot)$  and  $\text{PATH}(\cdot)$  are considered directional. Our proposed method is therefore applicable to directed graphs, where the symmetry condition does not hold.

TABLE 2  
Queries and techniques related to the continuous detour query

Technique	Optimization Goal	Temporality	Number of Results	Global Access
TPQ [13]	<i>Trip Distance</i>	Snapshot	<i>Multiple</i>	<i>No</i>
OSR [23]	<i>Trip Distance</i>	Snapshot	<i>Multiple</i>	<i>No</i>
OSR with AWVD [24]	<i>Trip Distance</i>	<i>Continuous</i>	Single	Yes
IRNN [27]	Deviation	Snapshot	<i>Multiple</i>	<i>No</i>
PNN Monitoring [4]	Deviation	<i>Continuous</i>	<i>Multiple</i>	<i>No</i>
Road Network $k$ NN [26]	<i>Trip Distance</i>	Snapshot	<i>Multiple</i>	<i>No</i>
Road Network Group $k$ NN [21]	<i>Trip Distance</i>	Snapshot	<i>Multiple</i>	Yes
$Ik$ SPT (Proposed)	<i>Trip Distance</i>	<i>Continuous</i>	<i>Multiple</i>	<i>No</i>

NN using the Voronoi diagram; (ii) deriving the subsequent NNs from neighboring Voronoi cells. Other precomputation-based techniques include  $k$ NN algorithms [7], [22] that use precomputed shortest path information stored in quadtrees and grid-based data structures.

Next we discuss continuous NN (CNN) problems in spatial networks. In this paper, we consider the setting of moving query objects and stationary data objects. We omit discussion on another class of CNN techniques [7], [15] which address problems in the setting of moving data objects.

Kolahdouzan and Shahabi [11] proposed the *upper bound algorithm (UBA)* for continuous  $k$ NN queries in a spatial network. The algorithm retrieves  $(k + 1)$  NNs with respect to a given location and calculates an upper bound. This upper bound is used to eliminate  $k$ NN computations between locations that  $k$ NN does not change.

Cho and Chung [5] proposed a continuous  $k$ NN technique that performs snapshot  $k$ NN queries at the intersections on the query path. They showed that  $k$ NN results between any two intersections can be inferred from those of the intersections. They also formulated an algorithm to find points where  $k$ NN changes for a predetermined query trajectory.

Nutanong et al. [16] proposed a technique called the  $V^*$ -diagram. In addition to the regular  $k$  NNs with respect to a given query point, their technique retrieves  $x$  auxiliary objects. The value of  $x$  is generally in the same order of magnitude as  $k$ . Retrieval of these auxiliary objects provides additional search scope to allow the query point to move while retaining enough information to continuously produce  $k$ NN results. As a result, the access cost is reduced.

These NN techniques are aimed at finding an object with the smallest distance with respect to a single query point. Using the NN query to solve the detour query problem (by assigning the starting location to the query point) may result in an impractical route, especially when the nearest object is in the opposite direction to the destination. For example, in Figure 1, the nearest restaurant with respect to  $q_s$  is  $c$ , which provides the TRIPDIST greater than those of  $a$  and  $b$ .

### 3.2 Route Planning Queries

Our  $k$ MDO monitoring problem can be categorized as a route planning problem. Specifically, we use the term *route planning* to refer to problems of finding a route through multiple destinations with respect to given routing requirements. Queries and techniques closely related to our problem are summarized

in Table 2, where each is categorized by: (i) **optimization goal**: the distance measure it aims to minimize; (ii) **temporality**: whether it produces snapshot or continuous results; (iii) **number of results**: the number of resultant objects produced/monitored; (iv) **global access**: whether it incurs access to all objects and nodes in the dataset. For example, to produce a  $k$ MDO result for one location, a global-access method examines all data objects and network nodes in the entire network.

The last row displays our proposed technique, *incremental order- $k$  shortest path tree ( $Ik$ SPT)*. To illustrate how our proposed technique fits into the existing literature, the table also shows the differences between existing techniques and ours. For example, TPQ [13] and OSR [23] are snapshot queries, while the continuous OSR monitoring method [24] can handle a single result at a time and requires access to all objects to construct a Voronoi diagram. The PNN monitoring technique [4] aims to minimize the deviation from a dynamically changing path instead of minimizing the TRIPDIST. In summary, none of these existing techniques can monitor multiple CDQ results in a spatial network. Detailed discussions of these queries and techniques are given as follows.

Li et al. [13] proposed the *trip planning query (TPQ)*. Given a list  $L$  of types of objects, the TPQ finds the shortest route that includes objects of those types in  $L$  with respect to given start and end points. For example, a user can be interested in visiting a post office and a gas station before going to work. The TPQ finds the shortest route to work that includes a post office and a gas station. A similar query, called the *optimal sequenced route (OSR) query*, was proposed by Sharifzadeh et al. [23]. Given a point  $q$  and a sequence  $S$  of object types, the OSR query finds the route that starts at  $q$  that orderly passes through object types in  $S$ , and minimizes the travelling distance. The main difference between OSR and TPQ is that OSR is ordered, while TPQ is not.

Sharifzadeh and Shahabi [24] presented a safe region-based solution to the OSR query using the *additively weighted Voronoi diagrams (AWVD)*. In an AWVD, each generator point  $p_i$  is the location of the first visited object of each possible route  $R_i$  and the associated weight  $w_i$  is calculated from the travelling distance between the first and the last objects. The boundary between the AWVD cells of two routes  $R_1 : (p_1, w_1)$  and  $R_2 : (p_2, w_2)$  is a hyperbolic curve,  $\|v - p_1\| + w_1 = \|v - p_2\| + w_2$ . That is,  $R_1$  and  $R_2$  provides the same total travelling distance with respect to the starting location. Since associated weights  $w_i$  can be any

positive value, this technique can also be applied to TPQ.

Yoo and Shekhar [27] proposed the *in-route nearest neighbor (IRNN)* query. The IRNN query finds a stopover that minimizes a deviation from a given path. Since a user can deviate from the path via only a node, this becomes a closest pair problem [6] between two sets of locations: the set of possible stopovers, and the set of nodes along the preferred path.

Chen et al. [4] studied the problem of *path nearest neighbor (PNN)* for a moving query point  $q$ . Their studies include algorithms to maintain the shortest path from  $q$  to a fixed destination; and to monitor  $k$  PNNs, i.e., the  $k$  objects with the minimum deviations from the shortest path.

Based on the INE and IER frameworks [20], Yiu et al. [26] proposed a spatial-network variant of the *aggregate kNN (AkNN) query* [19]. The *AkNN* query retrieves  $k$  objects with the smallest aggregate distances to a set of query points. Given a start  $q_s$  and destination  $q_e$  in an undirected graph, the aggregate distance is equivalent to TRIPDIST when the aggregate function is SUM and the query set is  $\{q_s, q_e\}$ . We adopt the principle of *AkNN* querying and the INE framework to formulate a comparator (in Section 6).

Safar [21] proposed an algorithm to find *group k nearest neighbors* in a spatial network. That is, a set of  $k$  objects that minimize the sum distances from a given query set. Similar to the VN<sup>3</sup> approach [10], the algorithm utilizes the network Voronoi diagram and precomputed distances.

## 4 PRELIMINARIES

### 4.1 Network Distance Calculation

This subsection provides background understanding of network distance calculations. Given two points  $x$  and  $y$  in a network/graph, Dijkstra [9] introduced an algorithm to calculate the shortest path from  $x$  to  $y$  based on the concept of best-first graph traversal. Specifically, to find the shortest path  $\text{PATH}(x, y)$ , the algorithm uses  $y$  as a reference point and incrementally examines surrounding nodes until  $x$  is covered, or vice versa. This algorithm is also known as “distance scan” since all nodes with distances smaller than that of  $x$  have to be visited. In addition, one may resume the computation from where it is terminated to obtain the shortest path from  $y$  to any node farther than  $x$ .

In our problem setting, since we have to compute the distances from multiple detour objects to a single destination  $q_e$ , we further explain the Dijkstra’s algorithm using Figure 2. To calculate  $\text{PATH}(n_3, q)$ , the algorithm incrementally discovers surrounding nodes according to their distances to  $q$ . In this example,  $n_5$  is discovered first with a distance of 1 unit. Next,  $n_4$ ,  $n_1$  and  $n_8$  are discovered with distances of 7 (via  $n_5$ ), 8 (via  $q$ ) and 8 units (via  $n_5$ ), respectively. The search halts when  $n_3$  is discovered.

Figure 2 also shows that each of the nodes involved in the search is associated with a label  $(q, d, n_h)$  where  $q$  denotes the reference point,  $d$  denotes the shortest distance, and  $n_h$  denotes the node from which the shortest distance is derived. The shortest path from  $n_3$  to  $q$  can be obtained by recursively traversing the next hop  $n_h$  in the label  $(q, d, n_h)$  until  $q$  is reached. In this case, we obtain  $\langle n_3, n_4, n_5, q \rangle$  as the shortest

path. As displayed in Figure 2, a *shortest path tree* (where the reference point  $q$  is the root) is formed by linking the next hop  $n_h$  of all nodes involved in the calculation.

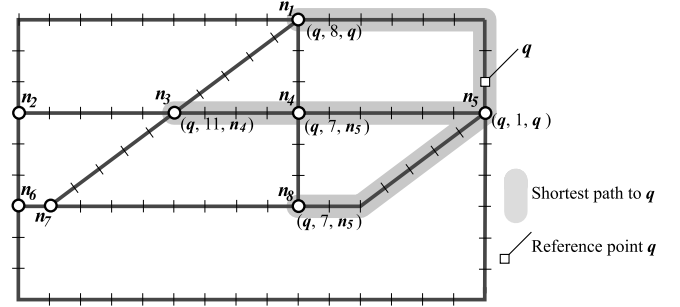


Fig. 2. End product of the Dijkstra’s algorithm where each node contains a label  $(q, d, n_h)$  denoting the reference point  $q$ , the distance  $d$  to  $q$ , and the node  $n_h$  from which the distance  $d$  is derived, respectively.

### 4.2 Voronoi Diagrams

This subsection provides discussion on variants of the Voronoi diagram as background knowledge for our proposed method. The most basic form of Voronoi diagrams [17] is the Voronoi diagram (VD) in Euclidean space with the order value of 1. Figure 3 shows the VD of 6 data objects as generators in Euclidean space (the underlying network connectivity is ignored). Each generator is associated with a cell in which the generator dominates. For example, the Voronoi cell  $\text{VC}(f)$  is a region containing points  $v$  such that the distance from  $v$  to  $f$  is not greater than the distance from  $v$  to any other generator. The boundary between two Voronoi cells  $p_i$  and  $p_j$  is defined by the bisector  $\text{BT}(p_i, p_j)$ , i.e., a set of locations equidistant to  $p_i$  and  $p_j$ . For example, the boundary of  $\text{VC}(f)$  consists of the bisectors  $\text{BT}(f, a)$ ,  $\text{BT}(f, c)$ ,  $\text{BT}(f, d)$  and  $\text{BT}(f, e)$ . This technique allows the NN problem to be treated in the same manner as a point-location problem. That is, locating the Voronoi cell in which the query point belongs.

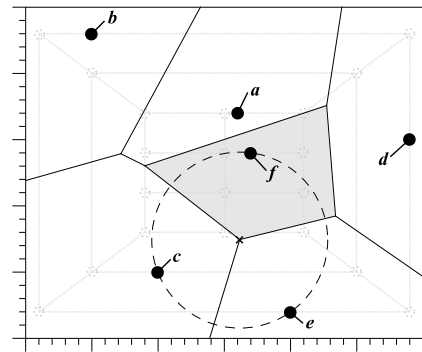


Fig. 3. Voronoi diagram in Euclidean space

Figure 4 displays a technique to construct a *network<sup>2</sup> VD*, where three boundaries (bisectors) of a sample cell  $\text{VC}(f)$  are marked as crosses. A network VD can be constructed using

2. The qualifier “network” is omitted when context is clear.

the *shortest path tree (SPT)* technique [18]. The technique is based on the best-first search principles similar to that of the Dijkstra’s algorithm [9]. Specifically, by replacing a single reference point with the objects (generators), we can use the Dijkstra’s algorithm explained in Section 4.1 to compute SPTs of multiple objects. An end product is a set of SPTs where each is associated with its nearest generator. Boundaries of Voronoi cells are obtained using the distance information embedded in each node to find points equidistant to two generators.

Figure 4 shows a forest of shortest path trees constructed based on the generators  $\{a, \dots, f\}$ . We consider this forest as a single tree when each branch is associated with its nearest generator and each generator is tied to a root. For example, the branch of generator  $a$  has  $\{n_4, \dots, n_7\}$ . The boundary between  $VC(a)$  and  $VC(f)$  on  $EDGE(n_5, n_8)$  is calculated based on  $DIST(n_5, a)$  and  $DIST(n_8, f)$ , which are 7 and 8 respectively. Therefore, the point on  $EDGE(n_5, n_8)$  that makes  $a$  equidistant to  $f$  is  $PNT(n_5, n_8, 2)$ , the point 2 units into  $EDGE(n_5, n_8)$  (measured from  $n_5$ ). This point is shown as the cross on  $EDGE(n_5, n_8)$ . A Voronoi cell can be obtained by applying this calculation to edges adjacent to its generator and the surrounding generators. For example  $VC(f)$  (the region enclosed by dotted lines) is bounded by the boundaries between  $f$  and its three neighboring generators  $a$ ,  $c$  and  $e$ . This technique can be used to provide answers for NN queries in a spatial network. For example, given a VD and an SPT for restaurants in a road network, the nearest restaurant can be obtained by locating the Voronoi cell that contains the user’s location, while the SPT provides the shortest route to that restaurant.

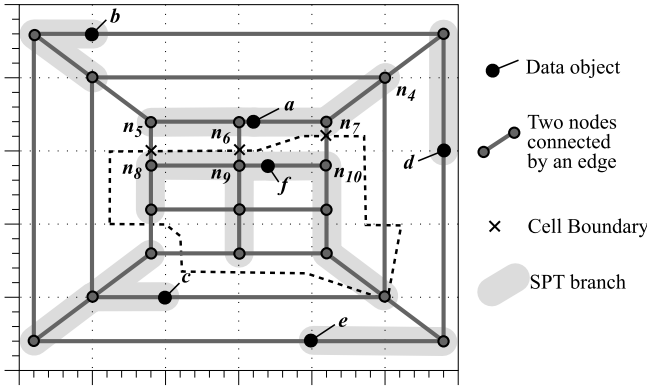
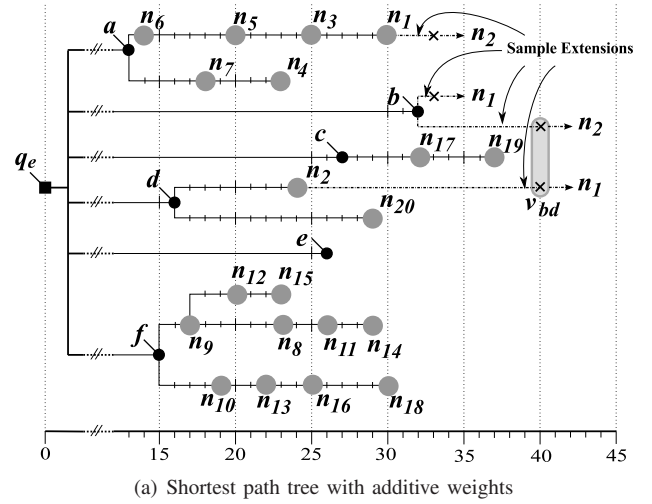


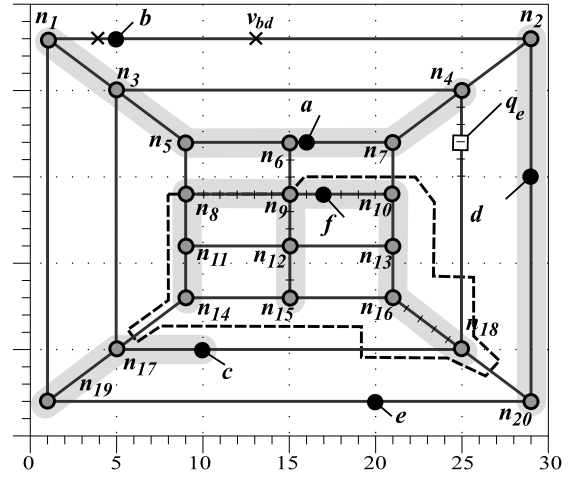
Fig. 4. Shortest path trees and a Voronoi cell

In the previous example, we assume that the preference for a particular restaurant is decided by the distance solely. The *additively-weighted VD (AWVD)* [18] is devised to address NN problems with an offset measure. For example, restaurants can be offset by their food prices to increase the desirability of cheaper restaurants. In the CDQ problem, each data object is associated with an offset derived from the distance to one common destination. However, the AWVD construction technique [18] is inapplicable to the CDQ problem because it only handles a single resultant object, and requires global access to network nodes and data objects. This technique forms the basis to describe our proposed method in the next section.

Figure 5 provides an example of the SPT technique using the additive weight concept. Figure 5(a) presents the order in which the nodes are included in the SPT. The  $x$ -axis represents the TRIPDIST measure. The SPT has 6 branches, where each branch corresponds to each of the six generators  $a$  to  $f$ . The initial position along the  $x$ -axis of an SPT branch corresponds to the additive weight (the distance from  $q_e$ ) of the corresponding generator. For example, the additive weight of  $a$  is 13 units so its initial position is 13. The first node to be included in SPT is  $n_6$  providing the TRIPDIST of 14. The SPT incrementally expands according to the additively weighted distance to the generators. The process stops when all nodes are labelled. In this example, the last node labelled is  $n_{19}$ .



(a) Shortest path tree with additive weights



(b) Additively weighted VD

Fig. 5. Construction of an AWVD using the SPT technique

After the SPT is constructed, the boundaries between Voronoi cells are calculated by finding edges  $EDGE(n_i, n_j)$  where  $n_i$  and  $n_j$  belong to two different generators. In Figure 5(b), the boundary between  $VC(b)$  and  $VC(d)$  on  $EDGE(n_1, n_2)$  is the point location  $v_{bd}$  such that  $TRIPDIST(v_{bd}, b, q_e)$  is equal to  $TRIPDIST(v_{bd}, d, q_e)$ . In this case,  $v_{bd}$  is  $PNT(n_1, n_2, 12)$ . This boundary is also shown as the extension to  $n_2$  on the branch of  $b$  and the extension to  $n_1$  on the branch of  $d$  in Figure 5(a).

## 5 PROPOSED SOLUTION

In this section, we present our proposed solution, *incremental order  $k$  shortest path tree ( $k$ SPT)* in the following steps. First, we generalize the SPT technique to the order- $k$  SPT ( $k$ SPT). Second, we show how  $k$ SPT construction can be done in an incremental manner. Finally, we present a method to compute the  $k$ MDO of any point in the network from a  $k$ SPT.

### 5.1 Order- $k$ Shortest Path Tree

In this subsection, we present a method to construct a  $k$ SPT by introducing overlaps between SPT branches. The  $k$ SPT branches are overlapped in such a way that each node appears in the tree exactly  $k$  times in  $k$  different branches. We first define the network data structure used by our proposed method. A network/graph is represented using the adjacency-list format, i.e., a list of nodes where each node entry contains information regarding its adjacent nodes. A data structure “Node” is defined as follows.

*Definition 3 (Node structure):* The structure of a node  $n_i$  contains the following attributes:

- *ID:* the node identification.
- *Adjacency list (AdjList):* a list of edges to/from immediate neighbors and associated weights. (For a directed graph, an adjacency list may comprise three edge types: incoming, outgoing and bidirectional.)
- *Label list (LabelList):* a list of (at most)  $k$  labels. For each label  $(p, d, n_h)$  in the label list of  $n_i$ ,
  - $p$  represents a detour object,
  - $d$  represents  $\text{TRIPDIST}(n_i, p, q_e)$ , and
  - $n_h$  represents the next hop in order to get to  $p$ .
- *Type:* a node type is “Labelable” by default and becomes “Permanent” upon completion of  $k$  labels.

Figure 6 shows a 2SPT of the network in Figure 5(b). The figure also shows the order in which nodes are processed according to the TRIPDIST measure. Each node in the 2SPT consists of two labels: *first* and *second*. For example,  $n_6$  appears twice in the 2SPT in the branches of generators  $a$  and  $f$ , respectively. The first label of  $n_6$  corresponds to the generator  $a$  and  $\text{TRIPDIST}(n_6, a, q_e)$  of 14 units. The second label of  $n_6$  corresponds to  $f$  and the TRIPDIST of 20 units. Note that the portion containing the first labels (highlighted in grey) is identical to the order-1 SPT shown in Figure 5(a).

We describe  $k$ SPT construction steps in Algorithm 1. The algorithm accepts a set  $\mathcal{D}$  of objects, a value of  $k$ , a graph  $G(N, E)$  and a destination  $q_e$  as input. The output  $k$ SPT is provided as  $G(N, E)$  with  $k$ MDO information embedded. Specifically, we introduce  $k$  labels for each  $n_i$  in  $N$ . The initialization (Lines 1 to 10) includes the following steps.

- First, a priority queue  $PQ$  is initialized. A priority-queue entry is a tuple  $(n, p, d, n_h)$ , where  $n$  is the node to which the entry corresponds, and the other three elements  $p$ ,  $d$ , and  $n_h$  form a labelling candidate for an entry in  $n.\text{LabelList}$ . Entries in  $PQ$  are ranked according to the labelling distance  $d$ .
- Second, for each object  $p$ , we create a node entry  $n_p$  and insert it into  $G(N, E)$  where affected edges in  $E$  are accordingly modified (Lines 3 and 4). We create a priority

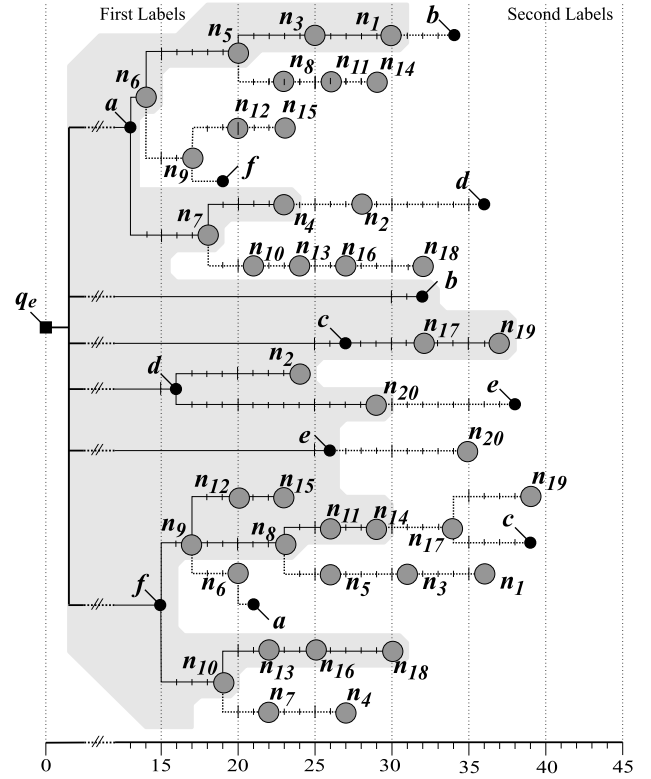


Fig. 6. Order- $k$  SPT with  $k$  of 2

queue entry for  $n_p$  with the associated detour object  $p$  and the labelling distance  $d$  of  $\text{DIST}(n_p, q_e)$ . The next hop  $n_h$  (shown as a dash “-” sign) is inapplicable in this case since  $n_p$  is already at  $p$ . The entry  $(n_p, p, d, -)$  is inserted in to  $PQ$  (Line 7).

- Third, for each network node, we create an empty label list and set the node type to “Labelable” (Lines 8 to 10).

Best-first search is handled by the *while loop* (Lines 11 to 20). The first step of each iteration is to dequeue the head entry  $(n, p, d, n_h)$  from  $PQ$  (Line 12). Since we are interested in only the first  $k$  labels of each node, the entry is ignored if the node’s label list already contains  $k$  labels, i.e., the node is “Permanent”. In addition, to ensure that each node is associated with  $k$  unique MDOs, the entry is also ignored if there exists an entry with Object  $p$  as the associated detour object in the label list. Otherwise, a label  $(p, d, n_h)$  is added to the node’s label list (Line 14). The node type becomes permanent if this label is the  $k$ -th entry in the label list (Lines 15 and 16). In Lines 17 to 20, for each node  $n_a$  such that there exists an edge that connects  $n_a$  to  $n$ , we create a  $PQ$  entry  $e_a, (n_a, p, d + \omega, n)$ , where

- $n_a$  is the node to which this entry corresponds;
- $p$  is the associated detour object;
- $(d + \omega)$  is the labelling distance calculated by adding the current labelling distance  $d$  to the weight of  $\text{EDGE}(n_a, n)$ ;
- $n$  is the node from which the labelling distance is derived.

The entry  $e_a$  is then inserted into  $PQ$ . The while loop continues until  $PQ$  is exhausted, i.e., every node is labelled  $k$  times. Finally, the graph  $G(N, E)$  with  $k$  labels on each node is returned as output (Line 21).

**Algorithm 1:** Construct- $k$ SPT( $\mathcal{D}$ ,  $k$ ,  $G(N, E)$ ,  $q_e$ )

---

**input** : Dataset  $\mathcal{D}$ ,  $k$ , Graph  $G(N, E)$ , Destination  $q_e$   
**output**: Labelled Graph  $G(N, E)$

---

```

1 Initialize Priority Queue  $PQ$ ;
2 for each (object  $p$  in  $\mathcal{D}$ ) do
3   Node  $n_p \leftarrow$  Create a network node from  $p$ ;
4    $G(N, E).Insert(n_p)$ ;
5   Distance  $d \leftarrow DIST(n_p, q_e)$ ;
6   PQEntry  $e \leftarrow$  Tuple  $(n_p, p, d, -)$ ;
7    $PQ.Insert(e)$ ;
8 for each ( $n$  in  $G(N, E)$ ) do
9    $n.LabelList \leftarrow$  Create an empty list of labels;
10   $n.Type \leftarrow$  Labelable;
11 while  $PQ$  is not empty do
12  PQEntry  $(n, p, d, n_h) \leftarrow PQ.DequeueHead()$ ;
13  if  $n.Type$  is Labelable and no existing label with  $p$  then
14     $n.LabelList.Add((p, d, n_h))$ ;
15    if  $n.LabelList.Length$  is  $k$  then
16       $n.Type \leftarrow$  Permanent;
17    for each Node  $n_a$  in  $n.AdjList$  that can immediately
18    reach  $n$  and  $n_a$  is Labelable do
19      Distance  $\omega \leftarrow EDGE(n_a, n).Weight$ ;
20      PQEntry  $e_a \leftarrow$  Tuple  $(n_a, p, d + \omega, n)$ ;
21       $PQ.Insert(e_a)$ ;
21 return  $G(N, E)$ ;

```

---

Let us now consider the first few steps of the algorithm, in the context of the example in Figure 6. After the initialization steps, the priority queue  $PQ$  has the following initial entries:

$$\langle (a, a, 13, -), (f, f, 15, -), (d, d, 16, -), \\ (e, e, 26, -), (c, c, 27, -), (b, b, 32, -) \rangle.$$

The first entry retrieved from  $PQ$  is  $(a, a, 13, -)$ . As a result, Node  $a$  is labelled with  $a$  itself as the associated detour object and the labelling distance of 13. Next, from the two nodes adjacent to  $a$ ,  $n_6$  and  $n_7$ , two entries  $(n_6, a, 14, a)$  and  $(n_7, a, 18, a)$  are created, respectively. These entries are inserted into  $PQ$  resulting in the following objects in  $PQ$ :

$$\langle (n_6, a, 14, a), (f, f, 15, -), (d, d, 16, -), \\ (n_7, a, 18, a), \dots, (b, b, 32, -) \rangle.$$

The second retrieved entry  $(n_6, a, 14, a)$  has  $n_6$  as the corresponding node, hence we apply the label  $(a, 14, a)$  to  $n_6$ . The same process continues until  $PQ$  is exhausted. The algorithm returns  $G(N, E)$  with  $k$  labels associated to each node as displayed in Table 3.

The drawback of Algorithm 1 is that it requires access to all data objects and nodes in the data space. This global access requirement can be disadvantageous especially in a large network. In the next subsection, we show how this drawback can be mitigated. To better illustrate this drawback in comparison to non-global access methods, we use Algorithm 1 as a competitor in the experimental studies.

TABLE 3

First and second labels of network nodes (with some entries omitted)

Node	First Label	Second Label
$a$	$(a, 13, -)$	$(f, 21, n_9)$
$b$	$(b, 32, -)$	$(a, 34, n_1)$
$c$	$(c, 27, -)$	$(f, 39, n_{17})$
$d$	$(d, 16, -)$	$(a, 36, n_2)$
$e$	$(e, 26, -)$	$(d, 38, n_{20})$
$f$	$(f, 15, -)$	$(a, 19, n_6)$
$\vdots$	$\vdots$	$\vdots$
$n_6$	$(a, 14, a)$	$(f, 20, n_9)$
$\vdots$	$\vdots$	$\vdots$
$n_{19}$	$(c, 37, n_{17})$	$(f, 39, n_{17})$
$n_{20}$	$(d, 29, d)$	$(e, 35, e)$

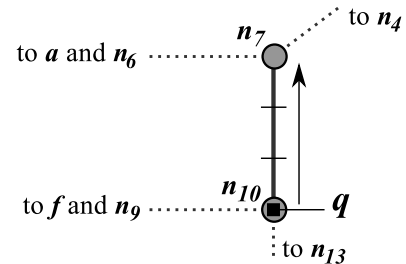
**5.2 Incremental  $k$ SPT Construction**

We now present our CDQ solution which incrementally retrieves data objects and computes node labels as the monitoring process progresses. The computation cost of a  $k$ SPT can be greatly reduced by exploiting the fact that the offset assigned to each object  $p$  is the distance from  $p$  to the destination  $q_e$ . Hence, objects that are far away from  $q_e$  are likely to be involved in the computation later than objects nearer to  $q_e$ . For example, in Figure 6, when applying the first label  $(a, 14, a)$  to  $n_6$ , Objects  $b$ ,  $c$  and  $e$  cannot affect the results since their distances to  $q_e$  are greater than the labelling distance of 14. Based on this property, we devise a mechanism which is incremental in two aspects:

- *Object retrieval*: through monitoring of the labelling distance and incremental retrieval of data objects;
- *Node labelling*: through an incremental labelling process, which halts when a desired label list is obtained and resumes when more label lists are required.

As a result, we eliminate the global access requirement in terms of both data objects and network nodes.

We revisit the running example in Figures 5 and 6 to elaborate the concept of incremental  $k$ SPT construction. As shown in Figure 7, the query point  $q$  is initially at  $n_{10}$  (and is travelling towards  $n_7$ ). The  $k$ MDO results of this initial location of  $q$  can be obtained from the label list of  $n_{10}$ .

Fig. 7. Starting location of the query point  $q$ 

We now describe how the label list of  $n_{10}$  can be obtained. As the labelling process progresses, detour objects are incrementally retrieved according to their distances to  $q_e$ . The scope of this object retrieval (the distance from the farthest retrieved object to  $q_e$ ) is denoted as a search radius  $r$ . The value of  $r$  indicates whether a node is safe to label or more detour

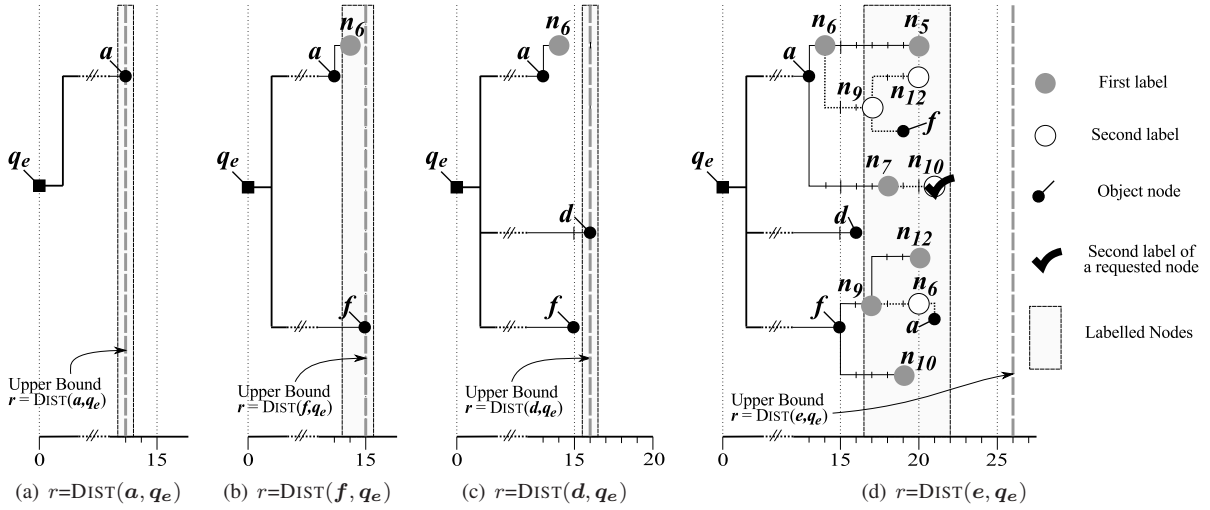


Fig. 8. Incremental  $k$ SPT construction with  $k$  of 2

objects are needed. Figure 8 presents a stepped explanation to how the  $k$  labels of  $n_{10}$  can be computed through incremental object retrieval and incremental node labelling.

- In Figure 8(a), the first retrieved object (the object nearest to  $q_e$ ) is  $a$ . The search radius  $r$  is set to  $\text{DIST}(a, q_e)$ . The only one node that can be labelled with this  $r$  value is Node  $a$  itself. After the labelling, we move on to consider the next object.
- In Figure 8(b), the next object nearest to  $q_e$  is  $f$ . The search radius  $r$  is updated to  $\text{DIST}(f, q_e)$ , which allows labelling of  $n_6$  and Node  $f$  itself, respectively.
- In Figure 8(c),  $d$  is retrieved and the value of  $r$  is updated  $\text{DIST}(d, q_e)$ . Node  $d$  is the only labelled node under this  $r$  value.
- In Figure 8(d),  $e$  is retrieved. The value of  $r$  is updated to  $\text{DIST}(e, q_e)$ , which allows the nodes in the grey region to be labelled. The labelling process halts upon completion of the second label of  $n_{10}$ . The figure also shows that  $n_{10}$  appears first in the branch of  $f$  and again in the branch of  $a$ . We can therefore infer that the  $k$ MDO list of  $n_{10}$  is  $\langle f, a \rangle$ .

Next, we show how a subsequent label list can be obtained as the query point moves away from the initial location in Figure 7. Assume that now the query point is at a location on  $\text{EDGE}(n_{10}, n_7)$ . In order to produce the  $k$ MDO results for this location, we need the label list of  $n_7$  in addition to that of  $n_{10}$ . Figure 9 shows that the second label of  $n_7$  can be obtained by resuming the labelling process halted after the completion of the second label of  $n_{10}$ . The figure also shows that we are only required to label  $n_{13}$  and  $n_7$  in order to obtain the second label of  $n_7$ .

Algorithm 2 provides detailed steps of how the label list of a node  $n_i$  is obtained. The first step is to check whether the  $k$  labels of  $n_i$  already exist, in which case the labels are returned right away (Line 1 to 3). For example, after obtaining the label list of  $n_7$  in Figure 9, the label lists of nodes  $n_6$ ,  $n_9$ ,  $n_{10}$  and  $n_{12}$  can be obtained without further graph traversal. If the requested label list is otherwise incomplete, we proceed to the main while loop (Lines 4 to 26). The main while loop

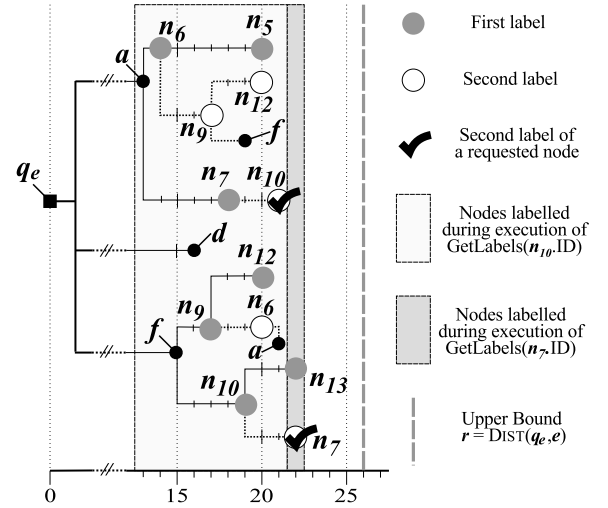


Fig. 9. Incremental  $k$ SPT construction with  $k$  of 2 (cont.)

in Algorithm 2 is similar to that in Algorithm 1. The following modifications are applied to make Algorithm 2 incremental.

- The first modification is the search radius check (Lines 6 to 12), which ensures that the value of  $r$  is not smaller than the labelling distance  $d$ . Specifically, until  $r$  is greater than or equal to  $d$ , the following steps are repeated:
  - (i) retrieving the next NN with respect to  $q_e$  (Line 7)<sup>3</sup>;
  - (ii) performing graph modification and priority queue insertion (Lines 8 to 12) similar to Algorithm 1;
  - (iii) setting the search radius  $r$  to the distance from that object to  $q_e$  (Line 12).
- The second modification is deferral of node initialization (Lines 15 to 17).
- The final modification is a halt to the node labelling process after the requested node has  $k$  labels (Lines 25 and 26).

3. We omit presentation of a marginal case where the dataset is exhausted, in which case we set  $r$  to infinity and break the loop.



**Algorithm 2:** GetLabelList(NodeID)

---

```

environment: Dataset  $\mathcal{D}$ ,  $k$ , Graph  $G(N, E)$ ,  $PQ$ ,  $r$ 
input       : NodeID
output      : LabelList  $\langle l_1, \dots, l_k \rangle$ 
1  $n \leftarrow \text{GetNode}(\text{NodeID}, G(N, E))$ ;
2 if  $n$  has been initialized and  $n$ .Type is Permanent then
3   return  $n$ .LabelList;
4 while  $PQ$  is not empty do
5   PQEntry  $(n, p, d, n_h) \leftarrow PQ$ .Head();
6   while  $r < d$  do
7     Object  $p \leftarrow \text{GetNextNN}(q_e, \mathcal{D})$ ;
8     Node  $n_p \leftarrow \text{Create a network node from } p$ ;
9      $G(N, E)$ .Insert( $n_p$ );
10    PQEntry  $e \leftarrow \text{Tuple}(n_p, p, \text{DIST}(p, q_e), -)$ ;
11     $PQ$ .Insert( $e$ );
12     $r \leftarrow \text{DIST}(p, q_e)$ 
13  PQEntry  $(n, p, d, n_h) \leftarrow PQ$ .DequeueHead();
14  if  $n$ .Type is Labelable and no existing label with  $p$  then
15    if  $n$  is not initialized then
16       $n$ .LabelList  $\leftarrow \text{Create an empty list of labels}$ ;
17       $n$ .Type  $\leftarrow \text{Labelable}$ ;
18     $n$ .LabelList.Add( $(p, d, n_h)$ );
19    for each Node  $n_a$  in  $n$ .AdjList that can immediately
    reach  $n$  and  $n_a$  is Labelable do
20      Distance  $\omega \leftarrow \text{EDGE}(n_a, n)$ .Weight;
21      PQEntry  $e_a \leftarrow \text{Tuple}(n_a, p, d + \omega, n_h)$ ;
22       $PQ$ .Insert( $e_a$ );
23    if  $n$ .LabelList.Length is  $k$  then
24       $n$ .Type  $\leftarrow \text{Permanent}$ ;
25      if  $n$ .ID = NodeID then
26        return  $n$ . $\langle L_1, \dots, L_k \rangle$ ;

```

---

**Algorithm 3:**  $Ik$ SPT-CDQ ( $\mathcal{D}$ ,  $k$ ,  $G(N, E)$ ,  $q_e$ )

---

```

environment: Trajectory  $T$  containing locations  $(n_i, n_j, \alpha)$ 
input       :  $\mathcal{D}$ ,  $k$ ,  $G(N, E)$ ,  $q_e$ 
output      : Reporting  $k$ MDO results at each location
1 Initialize Priority Queue  $PQ$ ;
2  $p \leftarrow \text{GetNN}(q_e, \mathcal{D})$ ;
3  $r \leftarrow \text{DIST}(p, q_e)$ ;
4 Node  $n_p \leftarrow \text{Create a network node from } p$ ;
5  $G(N, E)$ .Insert( $n_p$ );
6 PQEntry  $e \leftarrow \text{Tuple}(n_p, p, \text{DIST}(n_p, q_e), -)$ ;
7  $PQ$ .Insert( $e$ );
8 while More Location  $(n_i, n_j, \alpha)$  in  $T$  do
9   LabelList  $L_i \leftarrow \text{GetLabelList}(n_i, \text{ID})$ ;
10  LabelList  $L_j \leftarrow \text{GetLabelList}(n_j, \text{ID})$ ;
11  Results  $\mathcal{A} \leftarrow \text{GetResults}(\text{EDGE}(n_i, n_j), L_i, L_j, \alpha)$ ;
12  Report( $\mathcal{A}$ );

```

---

Algorithm 3 provides detailed steps of how  $k$ MDO monitoring can be conducted using Algorithm 2. Algorithm 3 has the following parameters: a dataset  $\mathcal{D}$ , a  $k$  value, a road-network graph  $G(N, E)$ , and a destination  $q_e$ . The initialization steps (Lines 1 to 9) include:

- initializing the priority queue  $PQ$ ;
- retrieving the nearest object to  $q_e$  as the initial object  $p$ , and inserting it into  $G(N, E)$  and  $PQ$ ;
- setting the search radius  $r$  to  $\text{DIST}(p, q_e)$ ;

The  $k$ MDO monitoring process is conducted in the while loop (Lines 9 to 12). In the context of our running example, Algorithm 3 uses Algorithm 2 to produce the label lists of  $n_{10}$  and  $n_7$ :  $\langle (f, 19, f), (a, 21, n_7) \rangle$  and  $\langle (a, 18, a), (f, 22, n_{10}) \rangle$ , respectively. These label lists are then used to produce the  $k$ MDO results for the current location of the query point (Line 11). These results are then reported to the user (Line 12). In the next subsection, we show how the  $k$ MDO of any location in  $G(N, E)$  can be derived from such label lists.

**5.3 Derivation of  $k$ MDO from Node Labels**

In the previous subsection, we have shown that the  $k$ MDO of any node in the network can be incrementally obtained using Algorithm 2. We now present a method to calculate the  $k$ MDO list of any location on an edge. For a query point  $q$  on a bidirectional edge  $\text{EDGE}(n_i, n_j)$ , the  $k$ MDO list of  $q$  may comprise objects from the lists of both end. (For a directional edge  $\text{EDGE}(n_i, n_j)$ , the  $k$ MDO list of  $q$  is the same as that of  $n_i$  if and only if  $q$  is exactly at  $n_i$ . Otherwise, the  $k$ MDO list is the same as that of  $n_j$ , since  $q$  cannot leave  $\text{EDGE}(n_i, n_j)$  without passing  $n_j$ .<sup>4</sup>)

We use Figure 10 to illustrate how the  $k$ MDO list can be obtained in such a case. Let  $\omega$  denote the edge's weight and  $\alpha$  denote the distance from  $n_i$  to the query point  $q$  along the edge. In this example, the query point  $q$  is on  $\text{EDGE}(n_{10}, n_7)$  with  $\alpha$  of 0.5 units and  $\omega$  of 3.0 units. The (order sensitive)  $k$ MDO lists of  $n_{10}$  and  $n_7$  are  $\langle f, a \rangle$  and  $\langle a, f \rangle$ , respectively. For each unique object  $p$  in those lists,  $\text{TRIPDIST}(q, p, q_e)$  can be obtained by comparing the distance via  $n_{10}$  and the distance via  $n_7$ . Specifically, we select the node that provides the minimum distance of  $(\text{TRIPDIST}(n_{10}, p, q_e) + \alpha)$  and  $(\text{TRIPDIST}(n_7, p, q_e) + \omega - \alpha)$ . Applying this principle to  $a$  and  $f$ , we can see that:

- $n_{10}$  provides the minimum distance of 19.5 units for  $f$ ;
- $n_7$  provides the minimum distance of 20.5 units for  $a$ .

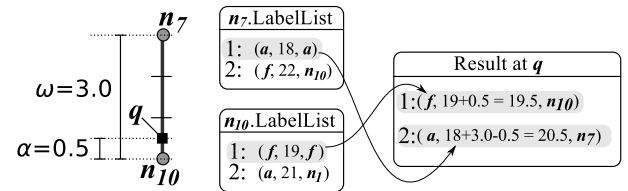


Fig. 10. Derivation of  $k$ MDO results from two label lists

Algorithm 4 provides a formal description of the result derivation process. The input parameters are (i) the edge  $\text{EDGE}(n_i, n_j)$  on which the query point resides, (ii) the two labels  $L_i$  and  $L_j$  of the end nodes, and (iii) the distance offset  $\alpha$  indicating the location of the query point relative to  $n_i$ . Lines 7 to 21 handle the case where  $\text{EDGE}(n_i, n_j)$  is bidirectional. (In Line 1, we check whether the edge is directional. If so, we return  $L_i$  if the query point is right on  $n_i$ . Otherwise,  $L_j$  is returned, since  $n_i$  is reachable only via  $n_j$  in this case.) For each unique object  $p$  from  $L_i$  and  $L_j$ , we compare the distance  $d_i$  via  $n_i$  and the distance  $d_j$  via  $n_j$ . If

4. Assume no objects on  $\text{EDGE}(n_i, n_j)$ .

there is no entry with  $p$  in  $L_i$ , the distance  $d_i$  is infinity (i.e., ignored). This is because in such a case, the actual  $d_i$  cannot affect the  $k$ MDO list of  $q$ . The same logic also applies to  $d_j$ . The minimum of  $d_i$  and  $d_j$  forms  $\text{TRIPDIST}(q, p, q_e)$ . Next, a label is created with the MDO of  $p$ , the labelling distance of  $\text{TRIPDIST}(q, p, q_e)$ . The label is inserted into the list  $\mathcal{A}$ . After the  $\text{TRIPDIST}$ s of all unique objects are calculated, the top  $k$  resultant labels in  $\mathcal{A}$  are returned.

---

**Algorithm 4:** GetResults(EDGE( $n_i, n_j$ ),  $L_i$ ,  $L_j$ ,  $\alpha$ )
 

---

```

input      : EDGE( $n_i, n_j$ ), LabelList  $L_i$ , LabelList  $L_j$ ,
              Distance  $\alpha$ 
output    : LabelList  $\langle l_1, \dots, l_k \rangle$ 
1 if EDGE( $n_i, n_j$ ) is directional then
2   if  $\alpha = 0$  then
3     return  $L_i$ 
4   else
5     return  $L_j$ 
6 else
7   Initialize a list  $\mathcal{A}$  of resultant labels;
8   for each unique object  $p$  from  $L_i$  and  $L_j$  do
9     Initialize  $d_i$  and  $d_j$  to infinity;
10    if there exists a label with  $p$  in  $L_i$  then
11       $(p, d, n_h) \leftarrow$  Label with  $p$  in  $L_i$ ;
12       $d_i \leftarrow d + \alpha$ ;
13    if there exists a label with  $p$  in  $L_j$  then
14       $(p, d, n_h) \leftarrow$  Label with  $p$  in  $L_j$ ;
15       $d_j \leftarrow d + \text{EDGE}(n_i, n_j).\text{Weight} - \alpha$ ;
16    if  $d_i \leq d_j$  then
17       $\mathcal{A}.\text{Insert}(p, d_i, n_i)$ ;
18    else
19       $\mathcal{A}.\text{Insert}(p, d_j, n_j)$ ;
20  Sort( $\mathcal{A}$ );
21  return Head( $k, \mathcal{A}$ )

```

---

The correctness of Algorithm 4 is guaranteed by Lemma 1. Specifically, the lemma states that the  $k$ MDO of a location between two nodes of a bidirectional edge can be obtained from the information given by the labels of those two nodes.

*Lemma 1:* Given an edge  $\text{EDGE}(n_i, n_j)$  and a common destination  $q_e$ , let  $\mathcal{A}_i$  and  $\mathcal{A}_j$  be the  $k$ MDO lists of  $n_i$  and  $n_j$ , respectively. If an object  $p$  is one of the  $k$ MDOs of any point  $v$  on  $\text{EDGE}(n_i, n_j)$ , then  $p$  is a member of  $\mathcal{A}_i$  and/or  $\mathcal{A}_j$ .

*Proof:* We prove this lemma by showing that for any object  $p$  that is neither a member of  $\mathcal{A}_i$  nor  $\mathcal{A}_j$ , there are at least  $k$  objects  $r$  in  $\mathcal{A}_i$  or  $\mathcal{A}_j$  such that

$$\text{TRIPDIST}(v, p, q_e) \geq \text{TRIPDIST}(v, r, q_e).$$

For any point  $v$  on  $\text{EDGE}(n_i, n_j)$ ,  $\text{TRIPDIST}(v, p, q_e)$  is the minimum between  $(\text{DIST}(v, n_i) + \text{TRIPDIST}(n_i, p, q_e))$  and  $(\text{DIST}(v, n_j) + \text{TRIPDIST}(n_j, p, q_e))$ . We therefore split derivation of  $\text{TRIPDIST}(v, p, q_e)$  into two cases. First, if  $n_i$  provides a  $\text{TRIPDIST}$  less than or equal to that of  $n_j$ , then

$$\text{TRIPDIST}(v, p, q_e) = \text{DIST}(v, n_i) + \text{TRIPDIST}(n_i, p, q_e).$$

Let  $p_i$  be any object in  $\mathcal{A}_i$ . If  $p$  is not in  $\mathcal{A}_i$ , then

$$\text{TRIPDIST}(n_i, p, q_e) \geq \text{TRIPDIST}(n_i, p_i, q_e).$$

This inequality implies that  $\text{TRIPDIST}(v, p, q_e)$  must be greater than or equal to that of any of the  $k$  objects in  $\mathcal{A}_i$ .

Second, when  $n_j$  provides a smaller  $\text{TRIPDIST}$ , the same principle can be applied to show that  $\text{TRIPDIST}(v, p, q_e)$  is not less than that of any of the  $k$  objects in  $\mathcal{A}_j$ .

These two cases imply that for any given point  $v$  on  $\text{EDGE}(n_i, n_j)$ , there must be at least  $k$  detour objects from either  $\mathcal{A}_i$  or  $\mathcal{A}_j$  that provide  $\text{TRIPDIST}$ s smaller than or equal to  $\text{TRIPDIST}(v, p, q_e)$ , and hence  $p$  can be safely ignored as a query result at  $v$ .  $\square$

Note that by considering data objects as network nodes, the  $k$ MDO results of any location on  $\text{EDGE}(n_i, n_j)$  can be derived from those of  $n_i$  and  $n_j$ . However, in a setting where such graph modification is inapplicable, the same principle can still be applied. In this case, to produce the  $k$  MDOs of any location  $q$  on  $\text{EDGE}(n_i, n_j)$ , we need to also consider detour objects along  $\text{EDGE}(n_i, n_j)$  in addition to the  $k$ MDO results at the end nodes.

We can observe that the  $k$ MDO of any location on an edge  $\text{EDGE}(n_i, n_j)$  can be efficiently computed by comparing the distances of entries in the label lists of  $n_i$  and  $n_j$ . Alternatively, one may apply a  $k$ NN monitoring algorithm (e.g., the split point calculation algorithm [5] or the incremental rank update algorithm [12]) to identify points along  $\text{EDGE}(n_i, n_j)$  where the  $k$ MDO changes. Using these split points, an edge can be decomposed into intervals where each corresponds to a particular  $k$ MDO. Consequently, we can replace repetitive distance comparison and sorting operations by boundary check and incremental update operations, respectively. Implementation of these techniques is independent of the proposed  $Ik$ SPT construction algorithm.

## 6 COMPETITIVE METHOD: INCREMENTAL NETWORK EXPANSION

In this section, we formulate a competitive method based on the existing concept of  $Ak$ NN querying [19], [26]. In this method, we apply the *multiple query method* [19] to retrieve  $k$  MDOs with respect to a starting location  $q_s$  and a destination  $q_e$ . We call this method INE-CDQ, since we use the *incremental network expansion (INE)* principle to retrieve

- (i) NNs from  $q_s$  (objects  $p$  that minimize  $\text{DIST}(q_s, p)$ );
- (ii) NNs to  $q_e$  (objects  $p$  that minimize  $\text{DIST}(p, q_e)$ ).

The order in which these objects are retrieved depends on the search coverage  $C_s$  with respect to  $q_s$  and the search coverage  $C_e$  with respect to  $q_e$ . That is, the next NN with respect to  $q_s$  is retrieved if  $C_s$  is smaller than  $C_e$  and vice versa. Based on these coverages, each retrieved object  $p$  is categorized into two types.

- *Candidate:*  $p$  is discovered via both  $q_s$  and  $q_e$ .
- *Pre-candidate:*  $p$  is discovered via either  $q_s$  or  $q_e$  only.

The  $\text{TRIPDIST}$  of each candidate  $p$  is the sum of  $\text{DIST}(q_s, p)$  and  $\text{DIST}(p, q_e)$ , while a lower bound is used to represent the smallest possible  $\text{TRIPDIST}$  produced by all pre-candidates.

Let  $p_s$  and  $p_e$  be the nearest pre-candidates discovered by  $q_s$  and  $q_e$  respectively. A  $\text{TRIPDIST}$  lower bound  $L$  of all pre-candidates is calculated by substituting the search coverages

$C_s$  and  $C_e$  for the unknown distances, i.e.,

$$L = \min\{\text{DIST}(\mathbf{q}_s, \mathbf{p}) + C_e, C_s + \text{DIST}(\mathbf{p}, \mathbf{q}_e)\}.$$

Assume that all candidates  $\mathbf{p}$  are maintained in a priority queue  $PQ_c$ , where entries are organized according to  $\text{TRIPDIST}(\mathbf{q}_s, \mathbf{p}, \mathbf{q}_e)$ . The head entry of  $PQ_c$  is the next MDO with respect to  $\mathbf{q}_s$  and  $\mathbf{q}_e$  if its  $\text{TRIPDIST}$  is smaller than the lower bound  $L$ . Otherwise, the next NN has to be retrieved in order to determine the next MDO. The process continues until  $k$  MDOs are discovered.

Since this method constructs  $k$ MDO results based on a specific query location, the results have to be reevaluated as the query point  $\mathbf{q}_s$  encounters an unvisited node. However, we can exploit the fact that  $\mathbf{q}_e$  is a fixed destination and reuse the network expansion result for subsequent  $k$ MDO evaluations. That is, all objects discovered via  $\mathbf{q}_e$  are cached and the SPT with respect to  $\mathbf{q}_e$  is incrementally expanded per demand. As a result, each node is visited by the network expansion with respect to  $\mathbf{q}_e$  at most once.

Note that one can also modify TPQ and OSR to find  $k$  MDOs with respect to a query location  $\mathbf{q}_s$  and a destination  $\mathbf{q}_e$  by limiting the sequence length to 2, where (i) the first element is a detour object  $\mathbf{p}$ , and (ii) the second element is a fixed destination  $\mathbf{q}_e$ . We can then find objects  $\mathbf{p}$  that minimize the total sequence length of  $(\text{DIST}(\mathbf{q}_s, \mathbf{p}) + \text{DIST}(\mathbf{p}, \mathbf{q}_e))$ . This modification results with an approach comparable to  $Ak$ NN.

## 7 DISCUSSION ON PROPOSED AND COMPETITIVE METHODS

In this section, we discuss the advantages and drawbacks of our proposed method  $Ik$ SPT-CDQ (Algorithm 3), in comparison to the following two competitive techniques.

- *INE-CDQ*: incremental network expansion around  $\mathbf{q}_s$  and  $\mathbf{q}_e$  (as described in Section 6.)
- *kSPT-CDQ*: construction of an entire order- $k$  shortest path tree (as shown in Algorithm 1.)

INE-CDQ applies the incremental network expansion principle to retrieve  $k$ MDO with respect to the current location of the query point  $\mathbf{q}_s$ . Since such an expansion is *query-centered*, the distances from  $\mathbf{q}_s$  to its surrounding nodes has to be reevaluated from scratch when  $\mathbf{q}_s$  moves. As shown in Algorithm 1 ( $k$ SPT-CDQ), we can avoid repetitive distance evaluation by applying network expansion in a *data-centered* manner. Specifically, network distances to a set  $\mathcal{D}$  of data objects are calculated by using each detour object  $\mathbf{p}$  in  $\mathcal{D}$  as a center of expansion (where  $\mathbf{p}$  is weighted by  $\text{DIST}(\mathbf{p}, \mathbf{q}_e)$ ). Algorithm 1 terminates only when each node  $\mathbf{n}$  is associated with  $k$  detour objects  $\mathbf{p}$  with the smallest  $\text{TRIPDIST}(\mathbf{n}, \mathbf{p}, \mathbf{q}_e)$ . As a result,  $k$ SPT-CDQ is disadvantageous to INE-CDQ when the query locations are close to the destination in relative to the size of the network.

In our proposed method,  $Ik$ SPT-CDQ, we exploit a special property of the CDQ problem where data objects are additively weighted based on their distances to the destination  $\mathbf{q}_e$ . As a result, the order in which nodes are visited by network expansion is skewed towards  $\mathbf{q}_e$ . Based on this knowledge,  $Ik$ SPT-CDQ is formulated by allowing the labelling process

to halt when the requested node is complete and to resume when more  $k$ MDO results are required. Therefore, the query processing cost depends on the distance from the query location  $\mathbf{q}_s$  to the destination  $\mathbf{q}_e$ . In this respect,  $Ik$ SPT-CDQ has a similar behavior to INE-CDQ. This is because, in order to retrieve the first MDO with respect to given locations of  $\mathbf{q}_s$  and  $\mathbf{q}_e$ , INE-CDQ has to consider at least objects in the search space of

$$\{\mathbf{v} : \text{DIST}(\mathbf{q}_s, \mathbf{v}) < d/2\} \cup \{\mathbf{v} : \text{DIST}(\mathbf{v}, \mathbf{q}_e) < d/2\},$$

where  $d$  denotes  $\text{DIST}(\mathbf{q}_s, \mathbf{q}_e)$ . However,  $Ik$ SPT-CDQ does not suffer from repetitive result evaluation, since  $k$ MDO results of each node are obtained via data-centric expansion. To provide a more comprehensive cost comparison between the three methods, experimental results are reported in the next section.

## 8 EXPERIMENTS

This section presents experimental studies on the two competitors, *INE-CDQ* and *kSPT-CDQ*, and our proposed method,  $Ik$ SPT-CDQ. Our experiments are conducted on a 2.66-GHz Intel Core 2 Duo machine with 4.0 GB of main memory. All algorithms are implemented in Java. As displayed in Figure 11, we use a large road network representing main roads in and around San Francisco. The network consists of 174,956 nodes and has a network diameter of 16,824 units.

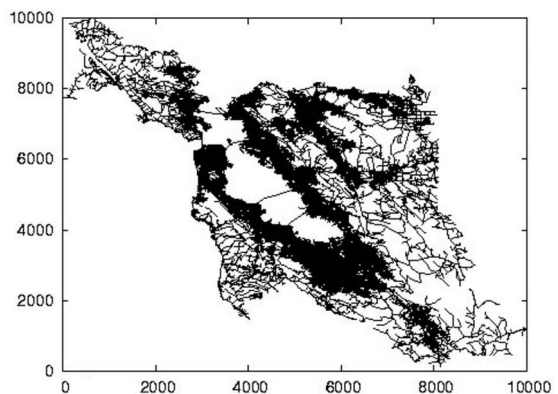


Fig. 11. Main roads around the San Francisco bay area

### 8.1 Experimental Setup

Our experimental setup is based on an application scenario where a user at the starting location  $\mathbf{q}_s$  wishes to travel to a location  $\mathbf{q}_e$ . The user also would like to monitor a list of  $k$  MDOs within a set  $\mathcal{D}$  of detour objects. After travelling for  $l$  units, the user selects a detour object and stops monitoring the  $k$  MDOs. Therefore, no further monitoring is needed and the query can terminate.

We use two performance measures: (i) the total execution time; (ii) the graph traversal cost. The total execution time is measured as the amount of time a technique uses to process one trajectory. The graph traversal cost is measured as the number of times network nodes are accessed through  $k$ SPT construction and network distance calculations.

Our experimental studies include the following parameters:

- the monitoring distance  $l$  along the query trajectory,
- the distance  $d$  from the original query location of  $q_s$  and the destination  $q_e$ ,
- the number  $k$  of requested MDOs, and
- the relative density  $\rho$  of nodes with respect to objects.

Table 4 provides the details of these parameters. The default values and ranges of these experimental parameters are derived from the aforementioned application scenario. We assume that a user (while moving) is capable of paying attention to only a small number of MDOs so the range of  $k$  is set to [2, 10]. The range of  $\rho$  is set to [100, 500] to emulate the number of nodes per point of interest (e.g., a restaurant or a hotel) that matches the user’s preference. That is, in a network with 174,956 nodes, the number of objects is ranged from 350 to 1750 objects. We use a high number of nodes per object, since data objects that match user’s preference are more realistically represented this way. For example, there maybe hundreds of restaurants of a particular cuisine and price range, although in total there maybe thousands of restaurant in one city.<sup>5</sup>

The range of  $d$  is set to [800, 4000] to represent typical travelling distances in one city, e.g., 5% to 25% of the diameter of the San Francisco network. The default  $k$ MDO monitoring distance  $l$  and the default value of the distance  $d$  from the origin to the destination are selected to emulate a user making their detour decision early in their trip. That is, the  $l$  value of 400 units and the distance  $d$  from  $q_s$  to  $q_e$  of 2,400 units correspond to the user making their detour decision after travelling 1/6 of the distance to the destination  $q_e$ . Assume that the user intends to drive to a destination 30 miles away from the origin and is travelling at a constant speed of 45 mph. The detour decision is made within the first 6-7 minutes of the trip.

TABLE 4  
Experimental parameters

Parameter	Default	Min	Max	Increment
$l$	400	0	800	200
$k$	6	2	10	2
$d$	2,400	800	2,400	800
$\rho$	300	100	500	100

We use two types of trajectories: *directional* and *random*. For each type, we generate 10 instances as a query set. Experimental results are reported as the average of results from these trajectories. We used the same set of starting locations  $q_s$  for both types. A directional trajectory was generated as the shortest path  $\text{PATH}(q_s, q_e)$  with a starting point  $q_s$ . The trajectory is terminated at the location  $v$  on  $\text{PATH}(q_s, q_e)$  such that  $\text{DIST}(q_s, v)$  is equal to  $l$ . A random trajectory was generated as a sequence of sub-trajectories (where the end of one sub-trajectory is the starting location of the next one). Each sub-trajectory is a shortest path to a random destination (independent of  $q_e$ ) and has a length of 100 units. The

5. Objects that do not match the user’s preference can be pruned during data retrieval (Line 7 of Algorithm 2 and Line 2 of Algorithm 3). Hence, they are not involved in  $k$ MDO labelling of nodes.

trajectory length  $l$  is measured as the total length of all sub-trajectories. For example, a trajectory with a length of 400 units contains 4 sub-trajectories.

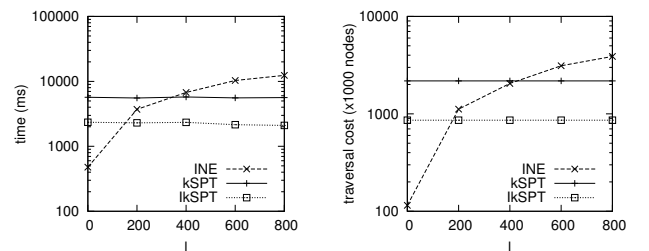
These two trajectory types represent two common cases. First, a directional trajectory represents a case where the directionality is absolute, e.g., a truck driver follows the shortest path to a destination while keeping track of a pizza place to get some food. Second, a random trajectory represents a case where the directionality is weak, e.g., a tourist visits sightseeing destinations while keeping track of supermarkets to drop by before going back to her hotel.

## 8.2 Experimental Results

### 8.2.1 Effect of $l$ on INE, $k$ SPT and $lk$ SPT

In the first set of experiments, we show the effect of the monitoring distance  $l$  on the execution time and traversal cost of the three methods. The value of  $l$  is ranged from 0 to 800 units. The  $l$  value of 0 corresponds to the query being used as a snapshot query, i.e., finding an instantaneous result without continuous monitoring. According to Figures 12 and 13, INE is the best method when used for a snapshot query. However, when the monitoring distance  $l$  increases, execution time and traversal cost of INE consistently increase for directional and random trajectories. This is because, INE has to reevaluate  $k$ MDO results for each unvisited node encountered by the query point.

For  $k$ SPT, on the other hand, both execution time and traversal cost remain unchanged, because  $k$ SPT computes the  $k$ MDO results for all nodes in the network. For  $lk$ SPT, changes in  $l$  do not produce a noticeable effect on both cost measures. This is because while computing the  $k$ MDO results for the initial query location of  $q_s$ ,  $lk$ SPT also computes results or partial results for its surrounding node. Hence, the incremental cost of computing subsequent node is negligible. The experimental results show that INE is the best method for a snapshot query, while  $lk$ SPT is the best method if continuous  $k$ MDO monitoring is required.



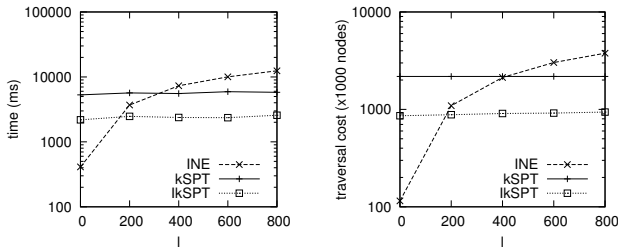
(a) Execution time (directional)

(b) Traversal cost (directional)

Fig. 12. Effect of  $l$  (directional trajectories)

### 8.2.2 Effect of $d$ on INE, $k$ SPT and $lk$ SPT

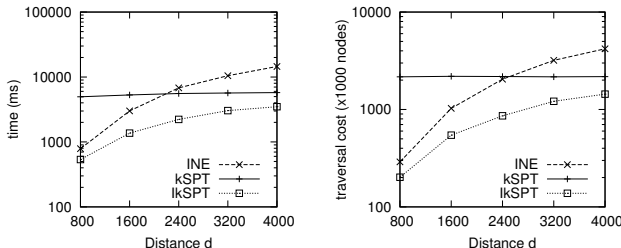
In the second set of experiments, we show how the three methods scale as the distance  $d$  from the initial location  $q_s$  to destination  $q_e$  increases. The distance  $d$  is varied from 800 to 4,000 units. According to Figures 14 and 15, since  $k$ SPT computes  $k$ MDO results for all nodes in the network,



(a) Execution time (random) (b) Traversal cost (random)

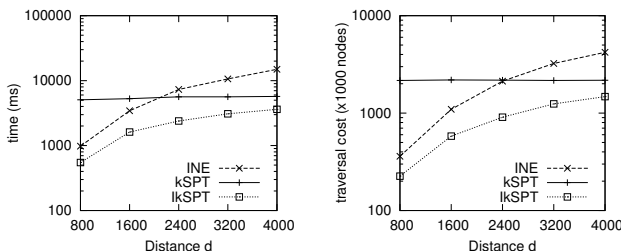
Fig. 13. Effect of  $l$  (random trajectories)

the distance  $d$  has no effect on either the execution time or the traversal cost for both directional and random trajectories. For INE and  $l$  $k$ SPT, on the other hand, the distance  $d$  has positive correlation with their execution time and traversal cost. These results conform with our discussion in Section 7. We can also see that  $l$  $k$ SPT has a much smaller cost and scales better than INE as  $d$  increases.



(a) Execution time (directional) (b) Traversal cost (directional)

Fig. 14. Effect of  $d$  (directional trajectories)



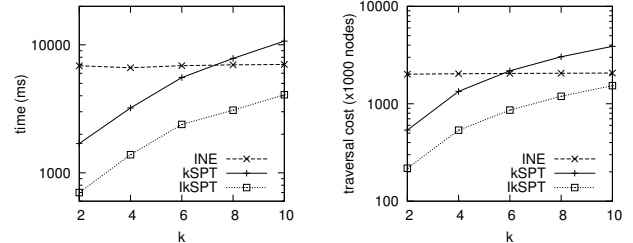
(a) Execution time (random) (b) Traversal cost (random)

Fig. 15. Effect of  $d$  (random trajectories)

### 8.2.3 Effect of $k$ on INE, $k$ SPT and $l$ $k$ SPT

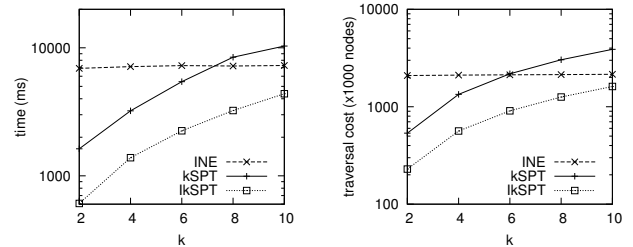
The third set of experiments presents a study on the effect of the number  $k$  of requested MDOs on the three methods. The value of  $k$  is varied from 2 to 10. According to Figure 16 (directional) and Figure 17 (random), the parameter  $k$  has no noticeable effect on the execution time or the traversal cost of INE. As explained in Section 7, each MDO has to be covered by the network expansions with respect to  $q_s$  and  $q_e$ . This requirement creates a high setup cost in order to determine the first MDO. This setup cost dominates the incremental cost of determining the subsequent  $(k - 1)$  MDOs. As  $k$  increases,

$k$ SPT and  $l$  $k$ SPT have similar behavior. That is, both execution time and traversal cost increase as  $k$  increase. This is because,  $k$  determines the number of labels for each node. Since  $k$ SPT has to construct the  $k$ MDO results for all nodes in the network and  $l$  $k$ SPT halts when requested nodes are complete, The execution time and traversal cost of  $l$  $k$ SPT are much smaller than those of  $k$ SPT.



(a) Execution time (directional) (b) Traversal cost (directional)

Fig. 16. Effect of  $k$  (directional trajectories)



(a) Execution time (random) (b) Traversal cost (random)

Fig. 17. Effect of  $k$  (random trajectories)

### 8.2.4 Effect of $\rho$ on INE, $k$ SPT and $l$ $k$ SPT

In the last set of experiments, we vary the parameter  $\rho$  from 100 to 500 nodes per object. According to Figures 18 and 19, for  $k$ SPT and  $l$  $k$ SPT, we see slight increases in the execution time and traversal cost as  $\rho$  increases. We have found that as the number of detour objects reduces (as a result of an increase in the number  $\rho$  of nodes per object), each  $k$ SPT branch (corresponding to a detour object) becomes larger and results in a greater number of priority queue entries. Hence, as  $\rho$  increases, we can see slight increases in the execution time and traversal cost of  $k$ SPT and  $l$  $k$ SPT for both directional and random trajectories (Figures 18 and 19). For INE, since the minimum search space is determined by the distance from  $q_s$  to  $q_e$ , there is no correlation between  $\rho$  and the traversal cost. The execution time of INE, on the other hand, decreases as  $\rho$  increases. This is because as  $\rho$  increases the number of objects reduces, which produces less candidates and pre-candidates to process in order to determine the  $k$  MDOs. We can also see that  $l$  $k$ SPT continues to outperform the other two methods in both cost measures.

## 8.3 Summary

We have derived ranges of experimental parameters and their default values based on a realistic CDQ setting. Since  $l$  $k$ SPT

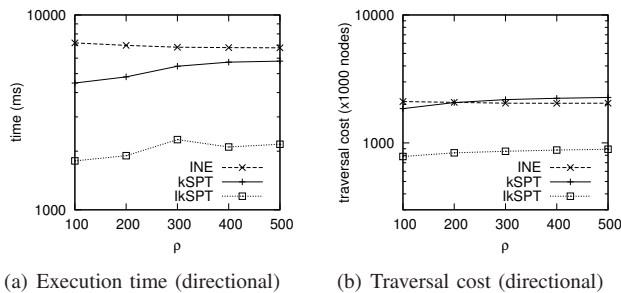


Fig. 18. Effect of  $\rho$  (directional trajectories)

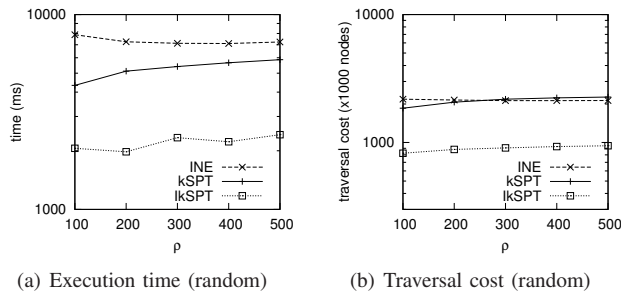


Fig. 19. Effect of  $\rho$  (random trajectories)

is a localization of  $k$ SPT,  $Ik$ SPT can only perform better than or similar to  $k$ SPT for both cost measures.  $Ik$ SPT scales better than INE as the distance  $d$  from  $q_s$  to  $q_e$  and the monitoring distance  $l$  increase. Although  $Ik$ SPT scales worse than INE as  $k$  increases,  $Ik$ SPT still performs better than INE for all values of  $k$  in the range of  $[2,10]$ .

## 9 CONCLUSIONS

We formulated the *continuous detour query* (CDQ) and proposed an efficient method to process the query. We have compared our proposed method,  $Ik$ SPT-CDQ, to two competitive techniques, INE-CDQ and  $k$ SPT-CDQ. INE-CDQ constructs  $k$ MDO results on a node-by-node basis and uses the Dijkstra’s algorithm [9] to explore detour objects around the current query location. When the query point encounters a new node, the  $k$ MDO results have to be reevaluated. The other competitive method,  $k$ SPT-CDQ, constructs  $k$ MDO results in a data-centered manner and calculates  $k$ MDO results for all nodes in the network.

Our proposed method,  $Ik$ SPT-CDQ, incrementally retrieves data objects according to their distances to the destination and computes  $k$ MDO results for a subset of nodes in the network. As a result,  $Ik$ SPT-CDQ does not incur access to all objects and network nodes. Since our method enables construction of  $k$ SPTs during runtime, one can apply preconditions to the object retrieval process in order to exclude irrelevant objects from  $Ik$ SPT construction.

Experimental results show that for the default values of parameters,  $Ik$ SPT-CDQ is 3.1 times as fast as INE-CDQ and 2.3 times as fast as  $k$ SPT-CDQ. In terms of the traversal cost,  $Ik$ SPT-CDQ has an improvement factor of 2.3 times and 2.4 times in comparison to INE-CDQ and  $k$ SPT-CDQ, respectively.

## REFERENCES

- [1] P. H. Bloch, N. M. Ridgway, and D. L. Sherrell. Extending the concept of shopping: An investigation of browsing activity. *Journal of the Academy of Marketing Science*, 17(1):13–21, 1989.
- [2] M. Brown, N. Pope, and K. Voges. Buying or browsing?: An exploration of shopping orientations and online purchase intention. *European Journal of Marketing*, 37(11):1666–1684, 2003.
- [3] M. A. Cheema, L. Brankovic, X. Lin, W. Zhang, and W. W. 0011. Multi-guarded safe zone: An effective technique to monitor moving circular range queries. In *ICDE*, pages 189–200, 2010.
- [4] Z. Chen, H. T. Shen, X. Zhou, and J. X. Yu. Monitoring path nearest neighbor in road networks. In *SIGMOD*, pages 591–602, 2009.
- [5] H.-J. Cho and C.-W. Chung. An efficient and scalable approach to CNN queries in a road network. In *VLDB*, pages 865–876, 2005.
- [6] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest pair queries in spatial databases. In *SIGMOD*, pages 189–200, 2000.
- [7] U. Demiryurek, F. B. Kashani, and C. Shahabi. Efficient continuous nearest neighbor query in spatial networks using Euclidean restriction. In *SSTD*, pages 25–43, 2009.
- [8] K. Deng, X. Zhou, H. T. Shen, S. W. Sadiq, and X. Li. Instance optimal query processing in spatial networks. *VLDB J.*, 18(3):675–693, 2009.
- [9] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.
- [10] M. R. Kolahdouzan and C. Shahabi. Voronoi-based  $k$  nearest neighbor search for spatial network databases. In *VLDB*, pages 840–851, 2004.
- [11] M. R. Kolahdouzan and C. Shahabi. Alternative solutions for continuous  $k$  nearest neighbor queries in spatial network databases. *GeoInformatica*, 9(4):321–341, 2005.
- [12] L. Kulik and E. Tanin. Incremental rank updates for moving query points. In *GIScience*, pages 251–268, 2006.
- [13] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S.-H. Teng. On trip planning queries in spatial databases. In *SSTD*, pages 273–290, 2005.
- [14] W. W. Moe. Buying, searching, or browsing: Differentiating between online shoppers using in-store navigational clickstream. *Journal of Consumer Psychology*, 13(1):pp. 29–39, 2003.
- [15] K. Mouratidis, M. L. Yiu, D. Papadias, and N. Mamoulis. Continuous nearest neighbor monitoring in road networks. In *VLDB*, pages 43–54, 2006.
- [16] S. Nutanong, R. Zhang, E. Tanin, and L. Kulik. Analysis and evaluation of  $V^*-k$ NN: An efficient algorithm for moving  $k$ NN queries. *To appear in VLDB J.*, 2010.
- [17] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. Wiley, Chichester, second edition, 2000.
- [18] A. Okabe, T. Satoh, T. Furuta, A. Suzuki, and K. Okano. Generalized network Voronoi diagrams: Concepts, computational methods, and applications. *International Journal of Geographical Information Science*, 22(9):965–994, 2008.
- [19] D. Papadias, Y. Tao, K. Mouratidis, and C. K. Hui. Aggregate nearest neighbor queries in spatial databases. *ACM Trans. Database Syst.*, 30(2):529–576, 2005.
- [20] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In *VLDB*, pages 802–813, 2003.
- [21] M. Safar. Group  $k$ -nearest neighbors queries in spatial network databases. *Journal of Geographical Systems*, 10(4):407–416, 2008.
- [22] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *SIGMOD*, pages 43–54, 2008.
- [23] M. Sharifzadeh, M. R. Kolahdouzan, and C. Shahabi. The optimal sequenced route query. *VLDB J.*, 17(4):765–787, 2008.
- [24] M. Sharifzadeh and C. Shahabi. Processing optimal sequenced route queries using Voronoi diagrams. *GeoInformatica*, 12(4):411–433, 2008.
- [25] R. W. White and D. Morris. Investigating the querying and browsing behavior of advanced search engine users. In *SIGIR*, pages 255–262, 2007.
- [26] M. L. Yiu, N. Mamoulis, and D. Papadias. Aggregate nearest neighbor queries in road networks. *IEEE Trans. Knowl. Data Eng.*, 17(6):820–833, 2005.
- [27] J. S. Yoo and S. Shekhar. In-route nearest neighbor queries. *GeoInformatica*, 9(2):117–137, 2005.
- [28] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee. Location-based spatial queries. In *SIGMOD*, pages 443–454, 2003.